SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation

Florence Maraninchi, Matthieu Moy, Jérôme Cornet, Laurent Maillet-Contoz, Claude Helmstetter, Claus Traulsen

SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation

Florence Maraninchi and Matthieu Moy
VERIMAG Laboratory
(UJF, CNRS, Grenoble INP)
firstname.lastname@imag.fr

Jérôme Cornet and Laurent Maillet-Contoz
STMicroelectronics
Grenoble, France
firstname.lastname@st.com

Claude Helmstetter
INRIA, LIAMA
Beijing, China
claude@liama.ia.ac.cn

Claus Traulsen
Kiel University, Germany
ctr@informatik.uni-kiel.de

***Abstract*— SystemC has become a de facto standard for the system-level description of systems-on-a-chip. SystemC/TLM is a library dedicated to transaction level modeling. It allows to define a virtual prototype of a hardware platform, on which the embedded software can be tested.**

Applying formal validation techniques to SystemC descriptions of SoCs requires that the semantics of the language be formalized. The model of time and concurrency underlying the SystemC definition is intermediate between pure synchrony and pure asynchrony.

We list the available solutions for the semantics of SystemC/TLM, and explain how to connect SystemC to existing formal validation tools.

I. Introduction

The Register Transfer Level (RTL) used to be the entry point of the design flow of hardware systems, including systems-on-a-chip (SoCs). However, the simulation environments for such models do not scale up well. Developing and debugging embedded software for these low level models before getting the physical chip from the factory is no longer possible at a reasonable cost. New abstraction levels, such as the *Transaction Level Model (TLM)* [1], have emerged. The TLM approach uses a component-based approach, in which hardware blocks are modules communicating with so-called *transactions*. The TLM models are used for early development of the embedded software, because the high level of abstraction allows a fast simulation. This new abstraction level requires that SoCs be described in some non-deterministic asynchronous way, with new synchronization mechanisms, quite different from the implicit synchronization of synchronous circuit descriptions.

SystemC is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to purely functional models. It comes with a simulation environment, and is becoming a *de facto* standard. SystemC offers a set of primitives for the description of parallel activities representing the physical parallelism of the hardware blocks. The TLM level of abstraction can be described with SystemC.

As TLM models appear first in the design flow, they become reference models for SoCs. Hence it becomes necessary to *validate* TLM models. However, the methods and tools that have been successful for the formal validation of circuits described at the RTL level (including model-checking techniques, methods based on SAT solvers, and methods based on theorem-provers) cannot be applied directly to TLM models. One reason is that TLM models do not have a simple synchronous semantics; another reason is that the language SystemC is defined on top of a general-purpose programming language (C++), which has no formal semantics definition.

In this paper, we investigate the problem of expressing the semantics of SystemC for TLM designs (denoted by SystemC/TLM in the sequel), in such a way that existing formal validation tools can be exploited. We concentrate on *model-based* validation techniques, such as model-checking and abstract interpretation. Formalizing SystemC for the use of theorem provers is outside the scope of this paper.

The paper is organized as follows. In section II we briefly explain what a TLM design is, and how to use SystemC for such descriptions. In section III we review the main approaches for the formalization of parallel and timed systems, among which we would like to find a candidate for the semantics of SystemC/TLM. Section IV reports on three experiments we made for the formalization of SystemC/TLM. Section V explains how the semantics can be implemented. Section VI is the conclusion. Related work is mentioned whenever needed.

II. Transaction-Level Modeling and SystemC

A. Example SystemC/TLM Design

A TLM model written in SystemC is based on an *architecture*, i.e. a set of components and connections between them. Components behave *in parallel*. Each component has typed connection *ports*, and its behavior is given by a set of communicating *processes* that can be programmed in full C++. For managing the set of concurrent processes that appear in the components, SystemC provides a *scheduler*, and several synchronization mechanisms: the low-level *events*, the synchronous *signals* that trigger an event when their value changes, and higher level, user-defined mechanisms based on abstract communication channels.

Figure 1 gives an example of a SystemC program. For clarity, we only show the body of the processes, and the methods called to process transactions in the slave modules. The system contains two master modules and two slave modules. They are connected through a `tac_router` channel (a TLM router channel developed in STMicroelectronics). The program also contains *assertions*, which shows one possible

way of expressing (safety) properties. The main program is not detailed here; it builds the architecture by instantiating components and communication channels. The *transaction* mechanism allows a process of a *master* module to call methods exported by *slave* modules.

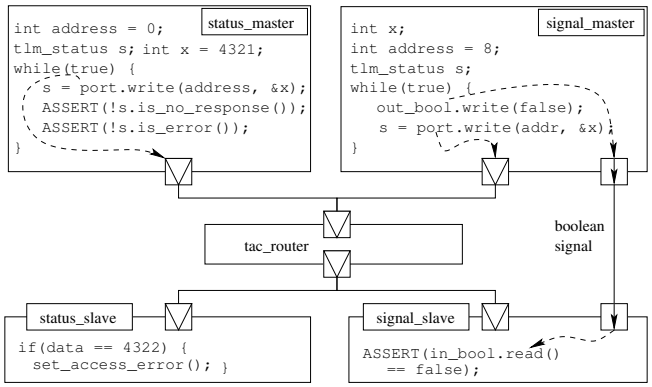

Fig. 1. An Example Transactional Model

B. The SystemC Scheduler

The SystemC Language Reference Manual [2] describes the scheduler algorithm. At the end of the initialization phase (construction of the platform by instantiating components and communication channels), some processes are *eligible*, some others are *waiting*. During the evaluation phase **EV**, eligible processes are run in an *unspecified order*, *non-preemptively*, and explicitly suspend themselves when reaching a *wait* instruction. A process may wait for some time to elapse, or for an event to occur. While running, it may access shared variables and signals, enable other processes by notifying events, or program delayed notifications. An eligible process cannot become "waiting" without being executed. When there is no more eligible process, signal values are updated (**UP**) and δ-delayed notifications are triggered, which can wake up processes. A delta-cycle is the duration between two update phases. Since there is no interaction between processes during the update phase, the order of the updates has no consequence. When there is still no eligible process at the end of an update phase, the scheduler lets time elapse (**TE**), and awakes the processes that have the earliest deadline. A notification of a SystemC event can be immediate, δ-delayed or time-delayed. Processes can thus become eligible at any of the three steps **EV**, **UP** or **TE**.

III. Models for Concurrent and Timed Systems

Formalizing the semantics of SystemC requires that we take into account: the processes written in C++, the behavior of the non-preemptive scheduler, and the notion of simulation time. We concentrate on the family of formal models that rely on the definition of *automata* to represent basic sequential activities, and on *products* to represent parallelism.

A. Basic Interpreted Automata

The basic elements are *interpreted automata*, or *extended automata*. These objects are made of a discrete and finite

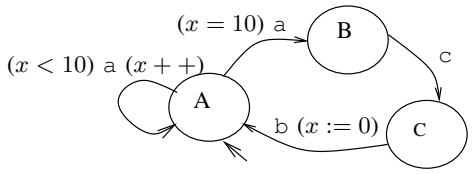

Fig. 2. An interpreted automaton with one numerical variable x.

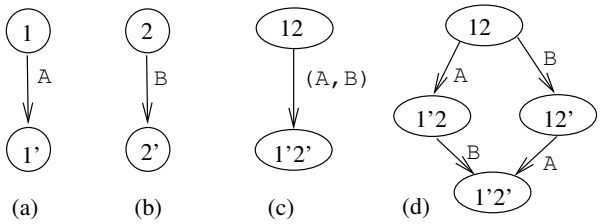

Fig. 3. Synchronous (c) and Asynchronous (d) Automata Products

control structure (states and transitions), plus a set of variables that can be of any type. A transition may test these variables, and assign new values to them. An interpreted automaton has the same expressive power as a Turing machine; any general-purpose sequential programming language can be encoded into such an automaton (this is more or less what a compiler does). Figure 2 is an example of such an automaton, in which `a`, `b` and `c` may represent external abstract events.

B. Synchronous and Asynchronous Products

Two kinds of automata products can be used to represent the parallel execution of two activities. The *synchronous product* is adequate for systems like synchronous circuits, in which the parallel activities share a common clock. The *Asynchronous* product is adequate for representing parallelism when no common clock exist between the parallel activities. This is often the case for systems made of several computers.

Figure 3 illustrates the two products. The two activities are described by finite automata with very abstract transition labels `A` and `B` (see (a), (b)). The *synchronous* product assumes a common clock: a transition of the product is made of a transition in each activity, because they "move" at the same time. This requires that we define the combination of labels (see (c)). The *pure asynchronous* product assumes no common clock; the system may evolve either because one of the activities moves, or because the other one does. There is no need for combining labels (see (d)).

C. Synchronization and Communication Mechanisms

The communication between synchronous activities may be *instantaneous*, because the parallel activities have instants in which they both move (see [3] for a full development on this subject). On the contrary, the communication between two asynchronous activities is necessarily asynchronous: since the two activities are never active at the same time, the only synchronization is based on shared memory.

Figure 4 illustrates the communication mechanisms in the two cases. In (a), the two automata will be composed synchronously. The labels may be structured as in

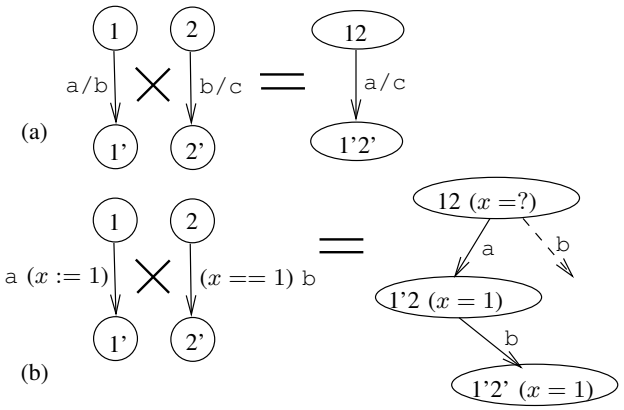

Fig. 4. Communication between Automata in the Synchronous case (a), and the Asynchronous case (b).

Mealy machines, with an input and an output (we note `input/output`). The product will force the transition `a/b` in one automaton to be executed "at the same time" (i.e., in the same transition of the product) as the transition `b/c` in the other automaton; this describes "instantaneous" communication via the signal `b`. In (b), the two automata will be composed asynchronously. The labels may explicitly refer to some common variable x, by assignments and tests. An assignment to x, in one automaton, can be observed by the other automaton in the following transitions.

D. Adding Discrete Time

In synchronous models, time is nothing more than an additional input (like `a` in Figure 4-(a)), and there may be several related time scales, like seconds, milliseconds, etc. In asynchronous models, time cannot be considered as an ordinary event, because it behaves in a very particular way: all the parallel entities of a system have the same "real" time: the two automata should execute their "time" transitions at the same time, which is not the normal effect of the asynchronous product. Adding time in the two forms of models described above can be done with explicit *clocks*, i.e., numerical variables representing the counting of time units. When time is discrete, these variables are very similar to the variables of the interpreted automata, as shown previously.

E. Existing Formalisms

There are a lot of formalisms for the description of parallel and timed activities. Most of them can be used as input for verification tools. Promela is the input language of the SPIN model-checker [4], and proposes an asynchronous product; the model-checker SMV [5] has an input language very similar to a synchronous composition of automata; the IF toolbox [6] allows to compose interpreted automata in an asynchronous way, and to express time as in timed-automata [7]; the Lustre toolbox [8] uses the synchronous programming language Lustre as the input language for a model-checker, an abstract-interpretation tool, a testing tool, etc.; Uppaal [9] is based on timed automata and asynchronous products, and has extensions for probabilistic and hybrid behaviors.

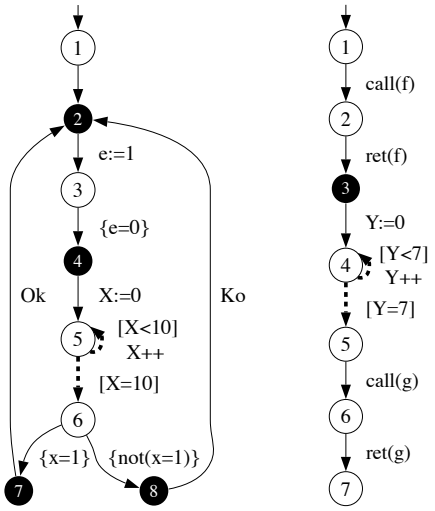

Fig. 5. Two micMac Automata

IV. Formalizing the Semantics of SystemC

The SystemC scheduler describes a situation which is neither purely synchronous, nor purely asynchronous. We experimented three ways of formalizing the semantics: 1) by an encoding into a synchronous formalism; 2) by an encoding into an asynchronous formalism; 3) by the definition of an ad-hoc product. For the sequel, we consider a SystemC/TLM model made of n processes $P_i, i \in [1, n]$; we ignore the module structure, since all the processes are scheduled by the same scheduler.

In the synchronous encoding, each process P_i is encoded into an interpreted automaton, equipped with special synchronizations intended to coordinate it with the scheduler; several automata that encode the scheduler are added; the scheduler automata prevent the process automata from executing all at the same time. The synchronous product of all these objects behave as the SystemC code. The code of functions is inlined in the callee.

In the asynchronous encoding, each process is encoded into an interpreted automaton, simpler than the one used for the synchronous case, and the scheduler is represented by a global shared variable that remembers which of the processes is active; the value 0 means no process is active, and the scheduler may choose any of the eligible ones. The code of functions is also inlined in the callee.

In the ad-hoc solution, each process, and each function body, is encoded into a so-called *micMac* automaton, and a special product is defined to represent all the details of the SystemC scheduler and the function call mechanism.

Figure 5 is an example micMac automaton. The first essential idea is to distinguish between *macro-states*, representing the points of the behavior of one process where the scheduler may indeed choose another process; and *micro-states*, representing the points where the process does not yield. The dedicated product considers that branching can be done at the macro-states (as in a classical asynchronous product) but not in the micro-states. The dedicated product will produce

branching between processes exactly when there is a choice in the scheduler.

The second idea is to encode time as it could be done in discrete timed automata. A micMac automaton has *timed* (represented by dashed arrows) and *untimed* transitions. Timed transitions may have conditions on variables that behave as the clocks of timed automata (for instance `[X<10]` on Figure 5). The dedicated product will ensure that time evolves in the same way for all the processes involved.

The third idea is to preserve the function call mechanism of SystemC, with dedicated labels like `call(g), ret(g)`. The dedicated product will match the body of a function with the process that calls it. We have to find statically an upper bound of the number of simultaneous callees of a function, to include as many copies of its body as necessary. This is feasible, assuming that,in SystemC designs, there is no recursion through communication function calls.

V. Implementation and Related Work

The implementation of all the proposals described above relies on the Pinapa [10] SystemC front-end, which parses the code of the processes with a C++ parser, and executes the elaboration phase in order to obtain a description of the architecture; it produces an internal representation from which all the semantic encodings can be performed.

The encoding into a synchronous formalism has been studied and fully implemented by M. Moy [11] using Pinapa, and with connections to several symbolic model-checkers and an abstract interpretation tool. To our knowledge, this is the only *complete* formalization of the SystemC semantics, and which is connected to a front-end. Other semantics have been proposed, but some of them are limited to the synchronous subset of SystemC (used for RTL designs, see [12] for instance); some others are very abstract TLM-like semantics, with no direct connections to a SystemC front-end.

In [13] we also investigated partial orders for the semantics of SystemC. The encoding into an asynchronous formalism has been described in [14] and experiments have been performed with Promela/SPIN; this involves manual abstractions between the SystemC code and the small automata of the model. The semantics using micMac automata and a dedicated product is fully described in J. Cornet's PhD [15].

The essential problem now is the size of the (implicit) automata produced by the full encoding of the semantics. The only hope for applying formal verification tools to SystemC/TLM designs of industrial size is to use very aggressive *abstractions* and/or *component-based* verification methods, that can deduce global properties of a system from local properties of its components. For the former, the manual abstractions experimented in the Promela modeling give hints for a more general, and automatic, method; one important point is how to identify the participants in a transaction, although this is specified by C++ `ints` representing addresses; since arithmetic on addresses is most of the time very simple, techniques from abstract interpretation should help a lot. For the latter, we need a component-based semantics of SystemC/TLM. This is not possible with our encodings into a synchronous or asycnhronous framework, because the scheduler is global. This is possible with the encoding into micMac automata plus the dedicated product.

VI. Conclusion

The complete chain between SystemC and symbolic model-checkers or abstract interpretation tools has demonstrated what we can do with SystemC/TLM designs. The development of the Pinapa front-end allows to take into account real case-studies from STMicrolectronics.

Further work will be done on TLM components, compositional verification, and the application of abstract interpretation techniques, in the context of the Minalogic/openTLM project at VERIMAG. On the other hand, the complete formal definition of the semantics can also be exploited for runtime verification, which is not exhaustive, but can accomodate more complex designs than exhaustive verification.

References

[1] F. Ghenassia, *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems*. Springer-Verlag, 2005.

[2] *IEEE 1666 Standard: SystemC Language Reference Manual*, Open SystemC Initiative, 2005, http://www.systemc.org/.

[3] F. Maraninchi and Y. Rémond, "Argos: an automaton-based synchronous language," *Computer Languages*, no. 27, pp. 61–92, 2001.

[4] G. J. Holzmann, *Design and validation of computer protocols*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[5] K. L. McMillan, "The SMV system, symbolic model checking - an approach," Carnegie Mellon University, Tech. Rep. CMU-CS-92-131, 1992.

[6] M. Bozga, S. Graf, and L. Mounier, "If-2.0: A validation environment for component-based real-time systems," in *Proceedings of CAV'02 (Copenhagen, Denmark)*, ser. LNCS, K. L. Ed Brinksma, Ed., vol. 2404. Springer-Verlag, July 2002, pp. 343–348.

[7] Alur and Dill, "The theory of timed automata," in *REX: Real-Time: Theory in Practice, REX Workshop*, 1991.

[8] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE," *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, Sept. 1992.

[9] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.

[10] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: The extraction tool for systemc descriptions of systems-on-a-chip," in *Fifth ACM International Conference on Embedded Software (EMSOFT)*, New-York, USA, Sept. 2005.

[11] ——, "LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level," *Design Automation for Embedded Systems*, vol. 10, no. 2-3, Sept. 2006, special issue on SystemC-based systems. [Online]. Available: http://www-verimag.imag.fr/ moy/publications/springer.pdf

[12] R. Drechsler and D. Große, "CheckSyC: An Efficient Property Checker for RTL SystemC Designs," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS 2005)*, vol. 4, May 2005, pp. 4167– 4170.

[13] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy, "Automatic generation of schedulings for improving the test coverage of systems-on-a-chip," in *6th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. San Jose, CA, USA: IEEE, Nov. 2006.

[14] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A systemc/tlm semantics in promela and its possible applications," in *SPIN Workshop*, 2007.

[15] J. Cornet, "Separation of functional and non-functional aspects in transactional level models of systems-on-chip," Grenoble INP Group," PhD, Apr. 2008.